Streaming SIMD Extensions--Diffuse Directional Lighting

Version 1.3

01/99

Order Number: 243630-004

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The Pentium® II processors, Deschutes processors, and Pentium® III processors may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Third-party brands and names are the property of their respective owners.

Copyright © Intel Corporation 1998

Table of Contents

1 Introduction 1
2 Illumination Function 1
2.1 Background Information 1
2.1.1 Benefits of SIMD Instructions 2
2.1.2 Efficient Cache Usage 2
2.2 Implementing the Illumination Function 2
2.2.1 Common Constants and Structures 2
2.2.2 Data Organization 3
2.2.3 Vertices Data Structures for AoS and SoA 4
3 Performance 5
4 Conclusion 5
5 Code Examples 6

Revision History

Revision	Revision History	Date
1.3	FCS version.	01/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

1 *Introduction to Computer Graphics*, Foley, van Dam, Feiner, Huges, and Philips, (Addison-Wesley, Reading, MA, 1993).

2 *Structures-of-arrays free 3-D bandwidth*, Julier and Hurley, *Electronic Engineering Times*, issue 955, May 26, 1997, pg. 96.

1 Introduction

Streaming SIMD Extensions for the Intel® Architecture (IA) instruction set provide floating-point single-instruction, multiple-data (SIMD) instructions. This application note discusses the use of Streaming SIMD Extensions to accelerate a 3D lighting operation. The example of a 3D lighting operation models the diffuse reflection of colored, directional light sources from colored surfaces. The Streaming SIMD Extensions provide performance improvement for this lighting operation through SIMD instructions that operate on packed, single precision, floating point data.

Other performance improvements can be obtained by data organization techniques, and improved cache usage. The code examples show two implementations of the lighting algorithm: one using the traditional Array of Structures (AoS) data arrangement, the other using a Structure of Arrays (SoA) arrangement.

The `\samples\clr_lit\` subdirectory contains C and assembly files with Streaming SIMD Extensions, and a small test program in C that you can build and run. The code examples use a directional lighting function based on the illumination equations discussed in *Introduction to Computer Graphics*.

2 Illumination Function

This section introduces the algorithm used to light and color a vertex in 3D graphics. It describes the illumination function used by the algorithm, and discusses two implementations using AoS and SoA data arrangements.

Directional light is a point light source at infinity, which produces the same intensity of light at every point on an object's surface. This algorithm uses an illumination function that calculates the diffuse reflection of colored directional light for each vertex of a colored surface. The illumination function for a single vertex is a vector dot product of the light source coordinates and the vertex normal, multiplied by the light color, the vertex material color, and the attenuation, as shown in Figure 1. In this example, the attenuation equals 1.0.

$$\cos\alpha = \begin{pmatrix} light.x \\ light.y \\ light.z \end{pmatrix} \bullet \begin{pmatrix} Vertex.Nx \\ Vertex.Ny \\ Vertex.Nz \end{pmatrix}$$

If $\cos\alpha > 0$ then:

$$\begin{pmatrix} vertex.r \\ vertex.g \\ vertex.b \end{pmatrix} = \cos\alpha \times \begin{pmatrix} light.r \\ light.g \\ light.b \end{pmatrix} \times \begin{pmatrix} material.r \\ material.g \\ material.b \end{pmatrix} \times attenuation$$

Figure 1: Illumination Function

2.1 Background Information

This section provides some background information on the main methods used for improving the performance of the illumination function. The main improvement is the result of using the SIMD feature of the Streaming SIMD Extensions, and better cache usage.

2.1.1 Benefits of SIMD Instructions

The single-instruction, multiple-data (SIMD) technique uses a single instruction to process multiple data elements simultaneously. Streaming SIMD Extensions contain SIMD instructions that operate on a packed single precision floating point (SP-FP) data type. Each packed SP-FP data operand contains four, 32-bit SP-FP data elements. Therefore, each instruction operates on four SP-FP data elements in parallel.

For example, a Streaming SIMD Extensions load operation loads four SP-FP numbers at once, whereas an conventional IA FP load operation loads only one floating-point number.

2.1.2 Efficient Cache Usage

A typical 3D algorithm transforms the 3D objects, and then applies lighting. Transformation is the process of putting the 3D objects in the correct location, with the correct size and angles. When using complicated 3D models, the size of the vertices data stored in memory typically exceeds the size of the L1 cache. For *N* light sources, the algorithm that transforms and lights the model makes *N*+1 passes on each vertex. Since the size of the vertices data in memory is larger than the L1 cache, the processor must load the vertices data *N* times from main memory or the L2 cache.

In general, both the transformation and lighting equations use the position of the vertices. Therefore, for an SoA data structure, these positions are in the cache after the transformation, so fewer cache misses occur when applying the lighting. The directional light function requires vertex normals, which are not used by the transformation algorithm. Therefore, cache misses still occur for each read of the SoA data structure.

To reduce memory traffic, the algorithm should perform as many calculations as possible on the data in the L1 cache, before the data is replaced by data for other vertices.

One approach is to perform all transformation and lighting calculations on a single vertex, before moving to the next vertex. However, this approach may require many function calls or branches per vertex, which adds significant overhead to the overall performance. If all pipeline functions are combined into one large function, using macros or inline functions for example, the calculation requires only one loop. This loop will be well-predicted.

Another approach is to do strip mining on a multi pass pipeline. In this application note, the vertices data is divided into blocks of 128 vertices. The transform and lighting calculations are performed one block at a time. This approach enables fetching data into the cache once, and all calculations are performed before replacing the data with new data.

2.2 Implementing the Illumination Function

This section describes how to efficiently implement the illumination function using the Streaming SIMD Extensions.

2.2.1 Common Constants and Structures

Both the AoS and the SoA implementations define these constants:

```
#define SIZE 128
float   atten = 1.0f;
```

where:

- `SIZE`
 defines the number of vertices processed in each block of data. Dividing the vertex data in blocks of 128 ensures efficient cache usage, as described in Section 2.1.2.
- `atten`
 specifies an attenuation factor used to calculate the light intensity at different distances from the light source. When the attenuation is set to 1.0, the intensity is regarded as the same everywhere, regardless of the distance from the light source.

Both implementations define a data structure, `_material`, which specifies the material properties:

```
typedef struct _material {
float r, g, b,// Material colors
reserv;         // Reserved for other properties of material
} Material;
```

In this definition, the material properties correspond to diffuse color. Other properties could be specified, such as ambient light, or specular light.

The AoS and SoA implementations also define a data structure, `_light`, which specifies the light properties:

```
typedef struct _light {
float x, y, z,// Light coordinates
   r, g, b,       // Light colors
   resa,       // Reserved for other properties of light
   resb;       // Reserved for other properties of light
} Light;
```

This structure specifies the color and position of the light only. Other properties could be specified, such as light direction, attenuation.

2.2.2 Data Organization

The way a data structure is declared determines how the structure's elements are stored in memory. This paper compares two ways of organizing the data for the vertices used in the lighting operation: AoS versus SoA. These data organizations are discussed in *Structures-of-arrays free 3-D bandwidth.*

AoS stores all the data elements for a single vertex sequentially in memory, as shown in Figure 2. This data arrangement is known as AoS format.

```
typedef struct _vertex {
   float x, y, z,
         nx, ny, nz,
         tu, tv;
} Vertex[N];
```

X	Y	Z	N_x	N_y	N_z	T_u	T_v
X	Y	Z	N_x	N_y	N_z	T_u	T_v

Figure 2: AoS Format

SoA stores the same data element for all vertices sequentially in memory, as illustrated in Figure 3. This data arrangement is known as SoA format.

```
typedef struct _vertex {
   float x[N],y[N],z[N],
         nx[N],ny[N],nz[N],
         tu[N],tv[N];
} Vertex;
```

X	X	X	X	X	X	X	X	···
Y	Y	Y	Y	Y	Y	Y	Y	···
Z	Z	Z	Z	Z	Z	Z	Z	···
N_x	N_x	N_x	N_x	N_x	N_x	N_x	N_x	···
N_y	N_y	N_y	N_y	N_y	N_y	N_y	N_y	···
N_z	N_z	N_z	N_z	N_z	N_z	N_z	N_z	···
T_u	T_u	T_u	T_u	T_u	T_u	T_u	T_u	···
T_v	T_v	T_v	T_v	T_v	T_v	T_v	T_v	···

Figure 3: SoA Format

Streaming SIMD Extensions process four data elements at a time, so the most efficient data organization is the SoA format.

2.2.3 Vertices Data Structures for AoS and SoA

The AoS implementation stores all data elements for a single vertex sequentially in memory. The SoA implementation stores the same data element for all vertices sequentially in memory. This section discusses both data structures in detail.

The AoS implementation defines the vertex data structure, as follows:

```
typedef struct _vertex {
   float x, y, z;      // Vertex's coordinates.
   float nx, ny, nz;   // Vertex's Normals.
   float tu, tv;       // Vertex's Texture coordinates.
} Vertex;
```

This structure is suitable for IA FP code, which loads only one data element at a time. However, Streaming SIMD Extensions load four data elements into each register. Thus, for the AoS implementation, each register contains three data elements, R, G and B, but the fourth element is unused.

To improve efficiency, the AoS algorithm presented in this application note loads the data in AoS format, then transforms the data to SoA format. In this way, subsequent lighting calculations are performed on four data elements simultaneously, rather than only three elements. Figure 4 illustrates the data layout in the xmm registers before and after the transformation.

Original AoS data

garbage	Nz_1	Ny_1	Nx_1
garbage	Nz_2	Ny_2	Nx_2
garbage	Nz_3	Ny_3	Nx_3
garbage	Nz_4	Ny_4	Nx_4

After Transpose

Nx_1	Nx_2	Nx_3	Nx_4
Ny_1	Ny_2	Ny_3	Ny_4
Nz_1	Nz_2	Nz_3	Nz_4

Figure 4: Transposing the Data

The SoA implementation defines the vertex data structure, as follows:

```
typedef struct _vertex {
   float x[SIZE],
           y[SIZE],   // Vertex's coordinates.
           z[SIZE],
           nx[SIZE],
           ny[SIZE],  // Vertex's Normals.
           nz[SIZE],
           tu[SIZE],  // Vertex's Texture coordinates.
           tv[SIZE];
} Vertex;
```

In this structure, each type of data element is stored in a separate array. For IA FP code, the SoA structure is equivalent to the AoS structure, since the code loads only one data element at a time. However, for Streaming SIMD Extensions, the SoA is preferable, since the load operation immediately loads four data elements. This eliminates the need to transform the data in the registers, as required in the AoS implementation.

3 Performance

The performance gains of the code are due to the following:

- Use of Streaming SIMD Extensions. These instructions are single-instruction, multiple-data (SIMD) instructions that can operate on four data elements at a time.
- The lighting algorithm is designed for efficient cache usage.
- SoA data arrangement improves performance compared to a traditional AoS data arrangement.

- The best performance is achieved by performing the lighting with data in the SoA format. In cases where the data is required in AoS format, it is still possible to use Streaming SIMD Extensions, by transposing the data to SoA format before the lighting operations, and writing the results to the appropriate place in the AoS output data structure.

4 Conclusion

Streaming SIMD Extensions can be used to accelerate 3D lighting functions, including diffuse reflection of colored, directional light from colored surfaces. The performance gains are due to:

- The SIMD nature of the Streaming SIMD Extensions, which operate on a packed single precision floating point data type.
- An algorithm that operates on data blocks containing 128 vertices, thereby enabling efficient cache usage.
- The advantage of using a SoA data arrangement for Streaming SIMD Extensions.

Performance of 3D lighting functions using Streaming SIMD Extensions is significantly improved over conventional IA floating-point code. Also, this code with data in SoA format performs better at the kernel level than the same code with data in AoS format.

5 Code Examples

The `\samples\clr_lit\` subdirectory contains the C and assembly code files, with the different implementations of the lighting function discussed in this application note:

`\soa\soa1.c`	SoA implementation in C code.
`\soa\soa2.asm`	SoA implementation in assembly language, with Streaming SIMD Extensions.
`\aos\aos1.c`	AoS implementation in C code.
`\aos\aos2.asm`	AoS implementation in assembly language, with Streaming SIMD Extensions.